UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Aplicación web para ayuda en el aprendizaje de la gestión de memoria dinámica en programación con el lenguaje C (continuación)

Óscar Martín Sanz
Tutor: Marina de la Cruz Echeandía
Ponente: Alfonso Ortega de la Puente

Julio 2018

Aplicación web para ayuda en el aprendizaje de la gestión de memoria dinámica en programación con el lenguaje C (continuación)

AUTOR: Óscar Martín Sanz
TUTOR: Marina de la Cruz Echeandía

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2018

Resumen

Este trabajo junto con el de Iván Serrano Sagredo, componen un producto completo, este documento contiene la segunda parte del proyecto. Éste se ha dividido en, parte web y generación de código, y en, interpretación de código y representación gráfica.

El objetivo de esta parte del proyecto es, dotar a la aplicación web de una herramienta para interpretar el código generado en la otra mitad del proyecto, y representar gráficamente cómo se desarrolla la memoria interna de un programa en ejecución escrito en C. El desarrollo de este producto ha sido diseñado para proporcionar una herramienta útil, fácil y compacta para las asignaturas de programación 2 de los grados de ingeniería informática y de telecomunicación de la *Escuela Politécnica Superior* de la *Universidad Autónoma de Madrid*. Proporciona la capacidad de, por un lado, generar código C bien formado, y por otro lado, ejecutar y depurar dicho código paso a paso visualizando gráficamente la evolución de la memoria y el valor de las variables del programa.

Ésta parte del proyecto abarca los bloques de la interpretación del código, la gestión de la memoria, y su representación gráfica en la web. Para ello se requiere que, la otra parte del proyecto, genere un código en C bien formado, ésto se hace por medio de bloques de Google Blockly. Una vez generado el código llega el momento de la interpretación, todo ello sin la necesidad de usar un servidor externo. En esta fase se ejecuta el programa. Se almacenan y gestionan todos los datos y posiciones necesarias para la ejecución del código y para la representación gráfica de cada paso de la depuración. Para el desarrollo del intérprete se ha diseñado y programado uno propio. Una vez se han guardado estos datos con éxito se llega a la representación gráfica, en la que se muestran la línea de código en ejecución, y las tablas con el contenido de las memorias. En estas tablas se muestran las direcciones, el tamaño, y el contenido de todas las variables reservadas de manera estática o dinámica, y de ámbito normal o local. Para el desarrollo del canvas gráfico nos hemos decantado por d3.js, ya que es una librería especializada en el manejo de datos.

Finalmente, si bien el uso del intérprete y de la representación gráfica fuera del proyecto común no ha sido contemplado, sí el uso del proyecto completo, junto con el de Iván Serrano Sagredo, para otros usos educativos dentro de otros ámbitos académicos.

Abstract

This work, with Ivan Serrano Sagredo's compose a complete product, this document contain the second part of the project. This project has been divided into the web part, the code generation and the code interpretation and graphic representation.

The main goal of this part of the project is to give to the web application a tool to interpret the code generated in the other half of the project, and represent graphically how the internal memory of an ejecución program written in C is developed. The development of this project has been design to provide an useful, easy and compact tool for "Programación 2" subjects of the informatic engineering and telecommunication degrees of the *Escuela Politécnica Superior* of the *Universidad Autónoma de Madrid*. Provided the capacity of, on the one hand, to generate a C code well formed, and, on the other hand, to run and refine this code step by step representing graphically the evolution of the memory and the value of the program variable.

This part of the project encompass the parts of the code interpretation, the memory management and the web graphic representation. For that it is required that the other art of the project generate a C code well formed, this is made up with blocks of Gloogle Blockly. Once the code is generated it is time to interpret, all of this without using and external server. In this part the program is run. All the data and the position needed to run the code and the graphic representation in each step of the refinement are stored and managed. To the interpreter development a new one has been design and programmed. Once this data has been successfully saved it is time for the graphic representation, where the code run line and the table with the memory contain are shown. In these tables the size, direction and content of all the reserved variables are shown in a static and dynamic way, and in a normal and local range. To the development of the canvas graphic we have chosen the d3.js, because it is a library specialised in data.

Finally, if it is true that the use of the interpreter and of the graphic representation outside the common project has not been contemplated, it has been in the use of the whole project with Ivan Serrano Sagrero's project for other educative uses within other academic fields.

Palabras clave

Blockly, bloques, intérprete, JavaScript, web, d3.js, vue.js, interpretar, canvas, memoria dinámica, C, educativo, análisis, gestión de memoria, funciones, condicionales, bucles, punteros, arrays, TADs, estructuras, workspace, if, else, for, while.

Keywords

Blockly, block, interpreter, JavaScript, web, d3.js, parser, canvas, Dynamic memory, C, educational, analysis, memory management, functions, conditional, loop, pointers, arrays, ADTs, structures, workspace, if, else, for, while.

Agradecimientos

Quiero agradecer su ayuda y apoyo a todas esas personas que nos han apoyado y acompañado a lo largo de nuestro paso por la carrera. A los amigos y compañeros con los que hemos compartido los últimos años.

Agradecérselo también a mis padres, pues ellos han supuesto un apoyo muy grande durante el día a día.

Gracias a los amigos del plan antiguo que me acogieron y ayudaron.

Gracias a los amigos que empezaron conmigo, aunque la mayoría no acabaran.

Gracias a los nuevos compañeros que me han regalado días de risas y de charlas interminables.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Prefacio

Este TFG se ha desarrollado desde un inicio como un trabajo único. De hecho, los dos alumnos involucrados han formado un único equipo de trabajo. Junto con el tutor y ponente han formado un único equipo que ha abordado partes del trabajo de manera conjunta aunque también se han repartido algunas tareas de manera independiente.

La manera de documentar el trabajo, desde nuestra perspectiva, más adecuada a la realidad del desarrollo y a la autoría de cada alumno, habría sido la elaboración de una memoria única de la que fueran autores ambos alumnos. Sin embargo, por requerimientos formales y legales de la normativa de los trabajos de fin de grado cada alumno debe presentar una memoria independiente.

Esto genera una situación un poco compleja respecto a la autoría de los textos que describen la parte común. La realidad es que han sido escritos por los dos alumnos como co autores y, para garantizar la coherencia de las memorias independientes deben aparecer en ambas; de otra forma resulten incompletas. Se ha intentado utilizar algún mecanismo de referencia de un documento al otro pero aún así quedan párrafos que es necesario que estén en las dos para que cada una de las memorias sea autocontenida. En ese sentido la memoria del trabajo de fin de grado de Iván Serrano Sagredo se citará como [20].

Tampoco pueden aparecer como autores de cada memoria los dos. La única manera en la que podemos explicitar que la coincidencia de ciertos textos en ambas memorias constituye la constatación de que ambos alumnos son los coautores de los dos es a través de este prefacio que esperemos sea suficiente para satisfacer tanto los requisitos formales de los trabajos de fin de grado como los de autoría de documentos públicos y ausencia de plagio en los mismos.

A continuación enumeramos los párrafos que aparecen en ambas memorias y de los que son autores ambos alumnos puesto que tanto la redacción como la revisión la han realizado de manera conjunta.

- Este prefacio.

- La introducción del trabajo.

- Párrafos previos a las figuras de la sección “Introducción” del “Estado del arte”

- En esa misma sección los textos finales desde la descripción de Snap! Y en las conclusiones de la misma.

- En el capítulo dedicado al diseño, las secciones iniciales hasta, e incluyendo, el ciclo de vida del desarrollo del proyecto.

- En el capítulo dedicado al desarrollo, el primer párrafo que describe la estructura del sitio web completo.

- En el capítulo dedicado a la integración y las pruebas, la sección de integración.

- En el capítulo de conclusiones y trabajo futuro, los tres primeros párrafos y el último de la sección de conclusiones.

- En ese mismo capítulo algunas frases de algunos párrafos de la sección de trabajo futuro.

Con este prefacio ambos alumnos manifiestan para que conste a todos los efectos su conformidad tanto en la autoría de los párrafos comunes como su presencia en ambas memorias en los términos que se ha intentado justificar en este prefacio.

2 Introducción

2.1 Motivación

En los últimos años hemos visto cómo la tecnología está mucho más presente en nuestra vida. Todos tenemos un teléfono móvil conectado a internet, donde podemos saber el tiempo que hará mañana o cuánto tardará el autobús en llegar a nuestra parada. Esta tecnología también ha permitido que el teletrabajo se vuelva algo mucho más común, algo que hace unos años era impensable. Recientemente hemos visto un aumento de dicha tecnología en los jóvenes estudiantes, y por ello muchos colegios, institutos y universidades han empezado a subirse a la ola tecnológica e implementar nuevas soluciones educativas lo que ha provocado una proliferación de herramientas web de carácter educativo, así como de librerías que permiten desarrollarlas de una manera mucho más sencilla, como es el caso de Blockly [1].

Gracias a las nuevas herramientas existentes, este proyecto nace con el objetivo de incrementar la competencia en la correcta gestión de la memoria dinámica en personas que o no tienen conocimientos, o se están introduciendo en el mundo de la programación, apoyándose para ello en las herramientas gráficas que permiten ayudar a imaginar visualmente el proceso de gestión de memoria dinámica mediante un formalismo de alto nivel distinto del lenguaje de programación C. Para ello, el proyecto consistirá en el desarrollo de una aplicación web que facilite la adquisición de la imagen visual, proporcionando un canvas en el que diseñar visualmente el proceso mediante el lenguaje de bloques Google Blockly, y un canvas gráfico en el que se representará la memoria del sistema, así como las modificaciones que las operaciones realice sobre ella. Así mismo, existirá un área de texto en la que se mostrará la equivalencia en lenguaje C de las operaciones diseñadas de manera visual.

2.2 Objetivos

Para la elaboración de este proyecto se definen los siguientes objetivos:

- Estudiar el estado del arte. (Común)
- Estudio del funcionamiento de Blockly (Común)
- Estudio de las herramientas de depuración de memoria. (Común)
- Estudio de las herramientas de diseño gráfico (Común)
- Desarrollo de un módulo Blockly con generador de código C que permita gestionar los programas típicos de las prácticas de la asignatura programación 2 del grado de ingeniería informática y telecomunicación. (Iván)
- Desarrollo de un módulo de representación gráfica de memoria. (Común)
- Desarrollo de un módulo que interprete el código C en JavaScript. (Óscar)
- Desarrollo de una aplicación web que integre todos los módulos. (Iván)
- Creación de una demo sobre el funcionamiento final de la aplicación. (Común)

Por razones de tiempo y simplificación se han omitido en el proyecto ciertas funcionalidades de C:

- Parámetros de entrada al programa.
- Entrada o salida estándar.
- Entrada o salida por fichero.
- Inclusión de librerías externas.
- Definición de cabeceras en módulo aparte.
- Enumeradores.
- Definiciones.
- Punteros a función
- Variables globales

2.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Capítulo 1**: Motivación, objetivos y organización de la memoria.
- **Capítulo 2**: Estado del arte de herramientas educativas enfocadas a la generación del código y herramientas de depuración de memoria.
- **Capítulo 3**: Diseño de la aplicación y sus módulos principales.
- **Capítulo 4**: Desarrollo de los módulos principales y sus componentes.
- **Capítulo 5**: Integración de los diferentes módulos y pruebas realizadas
- **Capítulo 6**: Conclusiones y trabajo futuro.
- **Apéndice A**: Manual de instalación.
- **Apéndice B**: Manual del programador.
- **Apéndice C**: Funciones del intérprete.

Al ser un TFG que junto con otro desarrolla un sistema único, cada uno ha desarrollado una memoria aparte. Oscar Martín se ha encargado principalmente del diseño, desarrollo y pruebas del intérprete, así como de la representación gráfica. Iván Serrano se ha encargado principalmente del diseño, desarrollo y pruebas de la Web y Blockly. En las siguientes tablas se detalla más cada parte:

	Diseño y análisis general	**Diseño y análisis Web**	**Diseño y análisis Blockly**	**Diseño y análisis Intérprete**	**Diseño y análisis Representación gráfica**
Iván Serrano Sagredo	X	X	X		X
Óscar Martín Sanz	X		X	X	X

	Desarrollo Web	**Desarrollo Blockly**	**Desarrollo Intérprete**	**Desarrollo Representación gráfica**
Iván Serrano Sagredo	X	X		
Óscar Martín Sanz			X	X

Tablas 2.3-1: División de trabajo

En base a esta distribución cada memoria contendrá una parte del análisis común y la parte de desarrollo única de cada trabajo. Para entender el funcionamiento por completo es necesario leer ambos documentos.

3 Estado del arte

3.1 Introducción

Existen diversas herramientas de lenguajes de bloques, como pueden ser *Blockly*, *Snap!* o Scratch, cada una de ellas con sus características. Tuvimos que considerar todas ellas para seleccionar la más adecuada a este proyecto.
Scratch, por ejemplo, es una herramienta enfocada a introducir a los usuarios (habitualmente estudiantes de ciclos anteriores a la universidad) a la programación y diseño de algoritmos. Por ello se ofrece un entorno de desarrollo enfocado en contar historias de forma interactiva o desarrollar videojuegos.
La acción se desarrolla en el canvas gráfico. El programador determina el comportamiento de todos los elementos dinámicos de la historia o el juego mediante algoritmos expresados con bloques. Todos los personajes se ejecutan en paralelo interactuando unos con otros habitualmente mediante eventos. El principal inconveniente es que es un paquete cerrado que dificulta el desarrollo de entornos más abiertos seleccionando sólo algunos elementos del paquete, como es nuestro caso.

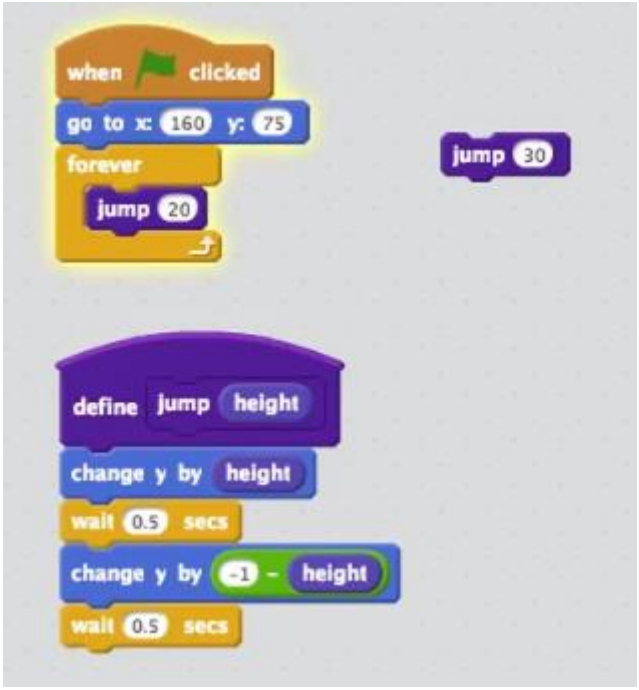

Figura 3-1: Scratch

La figura 2-1 muestra un ejemplo en código de bloques Scratch.

Snap!, por otro lado, es una herramienta orientada a un público más amplio. De hecho, realiza una tarea similar a Scratch (introducir al mundo de la programación) pero a estudiantes de mayor edad (se está utilizando en la universidad especialmente, aunque sólo en titulaciones distintas de informática). El sistema incluye un canvas gráfico similar al de Scratch. En apariencia parecen entornos con la misma funcionalidad, pero el código de Snap! es abierto y es JavaScript. El problema es que no está muy documentado y aunque

existen foros que comunican directamente con los autores, realmente es un espacio común para la continuidad de desarrolladores. Existe también un manual pero excluye los aspectos técnicos útiles cuando quieres incorporar sus elementos a un nuevo proyecto. No parece que los autores no tienen planes de facilitar esa documentación. De haber sido seleccionada para nuestro proyecto habríamos tenido que partir del código directamente.

Figura 3-2: Snap!

Por otro lado, Blockly [1] se ha convertido en un referente de hecho para la inclusión en cualquier sitio web de la capacidad de expresar un algoritmo mediante un lenguaje de bloques. Por ello se está utilizando mucho para la creación de herramientas educativas gráficas gracias a la facilidad y el acceso tan intuitivo que ofrece al usuario final. Existen multitud de proyectos que lo usan como base, como son **MIT App Inventor** [2], herramienta del *Massachusetts Institute of Technology* la cual abstrae de tal manera la creación de aplicaciones para dispositivos móviles que hasta niños son capaces de desarrollarlas, o **MakeCode**, desarrollada por Microsoft y sirve como introducción en la informática, así como cientos de proyectos más. En cuanto a proyectos enfocados a lenguajes de programación similares al descrito en esta memoria, existe un ejemplo desarrollado por Blockly [3] en el cual tratan la generación de código en lenguajes no tipados.

Figura 3-3: Proyectos en Blockly

En nuestro caso concreto, queremos introducir una herramienta educacional específica para el lenguaje C, más en concreto su gestión de memoria, donde tenemos a nuestra disposición una serie de herramientas que nos permiten ver desde diferentes puntos de vista el manejo de la memoria de un programa C. Estas herramientas otorgan una gran cantidad de información al usuario, tanto para controlar posibles errores durante una ejecución como para obtener información que permita mejorar el rendimiento de un programa.

3.2 Memoria

A lo largo de los años hemos observado cómo una de las partes que más complicada resultan de entender a la hora de introducirse en el mundo de la programación en C es realizar un manejo correcto de la memoria dinámica. Si bien tenemos a nuestra disposición herramientas como **Valgrind** [5], la cual nos permite depurar nuestro programa frente a errores o fugas en la memoria mediante comandos en terminal, **KCachegrind** [6], herramienta que usa los datos generados por Valgrind para representarlos gráficamente, o **DDD** [7], similar a KCachegrind pero tomando el *debugger* **GDB** [8] como fuente de datos, no existe ninguna herramienta, tanto online como local, que permita analizar el resultado de la memoria de un programa sin llegar a ejecutarlo.
En la siguiente imagen se muestra una captura de **KCachegrind.**

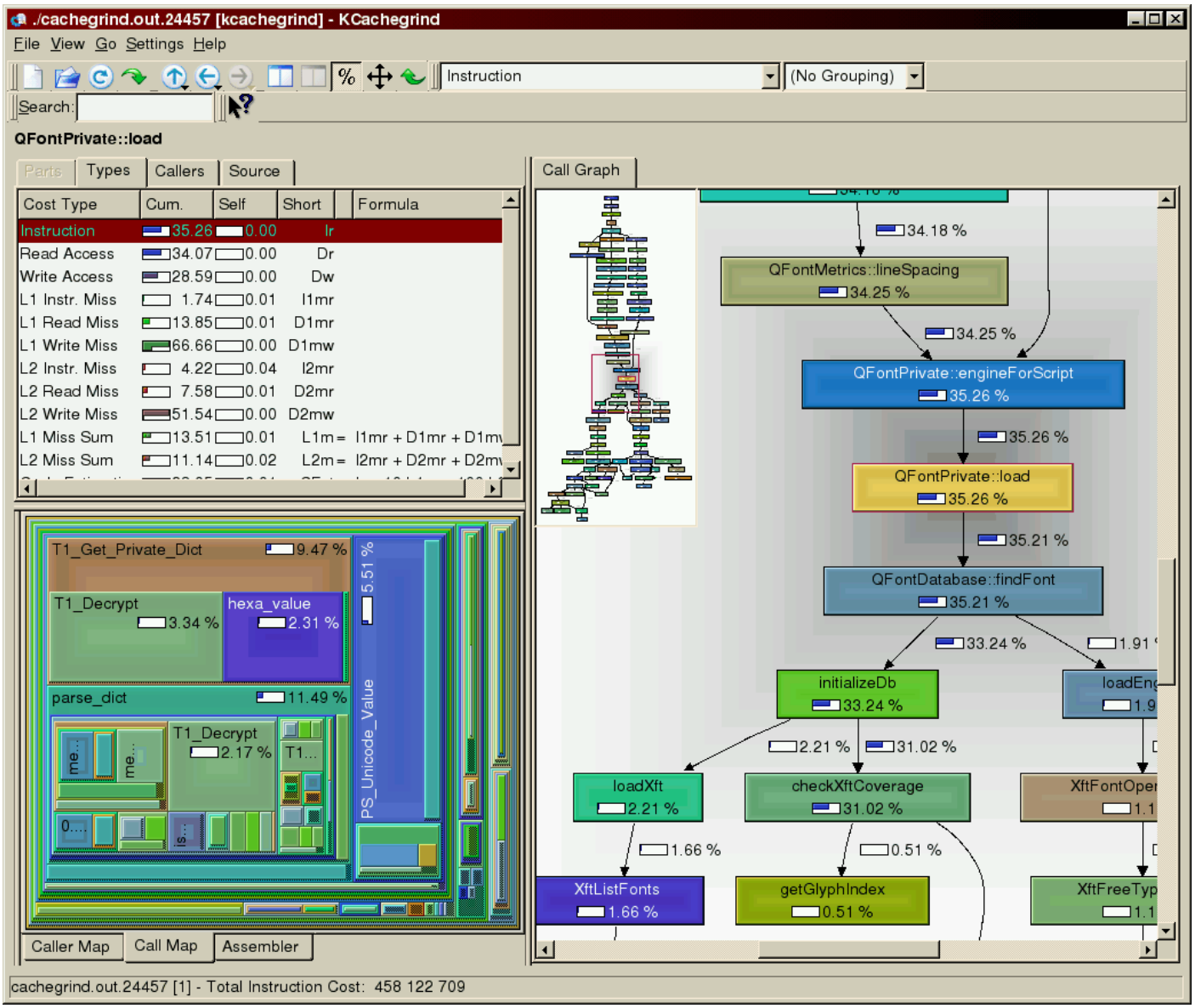

Figura 3-4: KCachegrind [6]

Existen intérpretes de C que permiten ejecutar un programa C como un script, como **Ch** [9] u otros **Tiny C Compiler** [10] como destinados a aliviar la carga de C para poder ejecutarlo y/o compilarlos en entornos con recursos limitados, como por ejemplo en el ámbito de la robótica. Sin embargo, ninguno de estos intérpretes se podían adaptar a nuestro caso de uso, ya que nuestro objetivo es realizar una plataforma web de aprendizaje, para lo cual necesitamos que esté desarrollado en un lenguaje web como JavaScript.
En la siguiente figura se muestra un ejemplo de DDD.

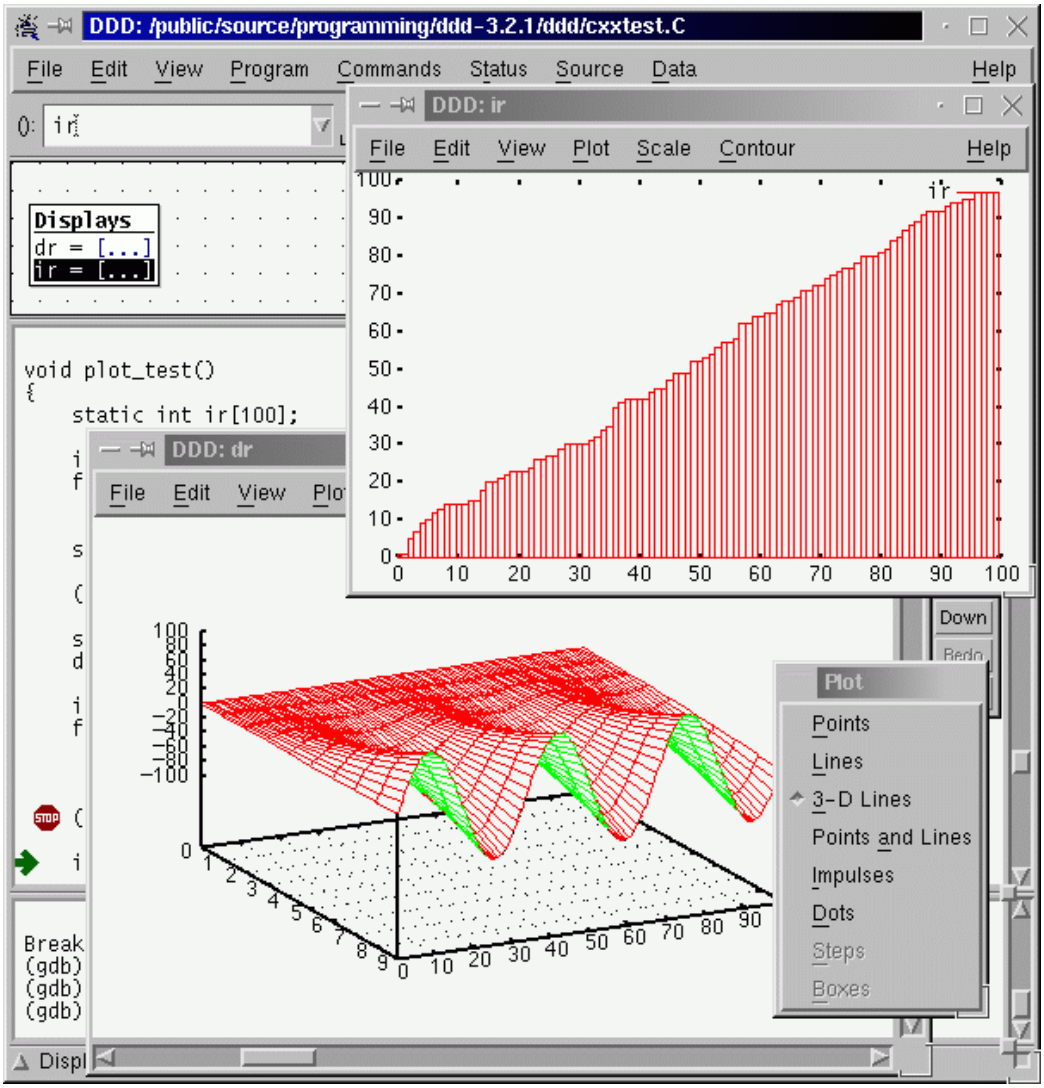

Figura 3-5: DDD [7]

3.3 Intérprete

A la hora de decidir qué hacer con el intérprete se nos plantean varias opciones, por un lado, surge la idea de buscar alguna herramienta o algún framework que solucione este problema.

Una de las primeras opciones que barajamos es el uso de la herramienta Jison [16] (cuyo logotipo está en la figura 2-6), ésta es una API basada en Flex y Bison para JavaScript, pero dado que lo que nosotros necesitábamos era una forma de preparar los datos para ser mostrados de forma gráfica, y no encontrar un módulo completo de Jison para C decidimos darle otra vuelta. Además del problema de que no bastaba con compilar el código y generar el código en ensamblador, sino que necesitábamos ejecutar ese código y almacenar el estado de todo el programa en cada caso de la ejecución. No hemos creído conveniente el uso de esta herramienta y se han buscado más opciones.

Figura 3-6: Jison

Otra opción que hemos contemplado para esta tarea es la API JavaScript-C, que es una versión de SpiderMonkey [17] de Mozilla, se trata de un intérprete para lenguajes orientados a objetos. Esta herramienta está pensada para ser usada con C++, y, pese a parecer la opción más indicada para ser elegida como la opción final, ha sido descartada por sus carencias en el trato de los punteros y de las estructuras de datos.

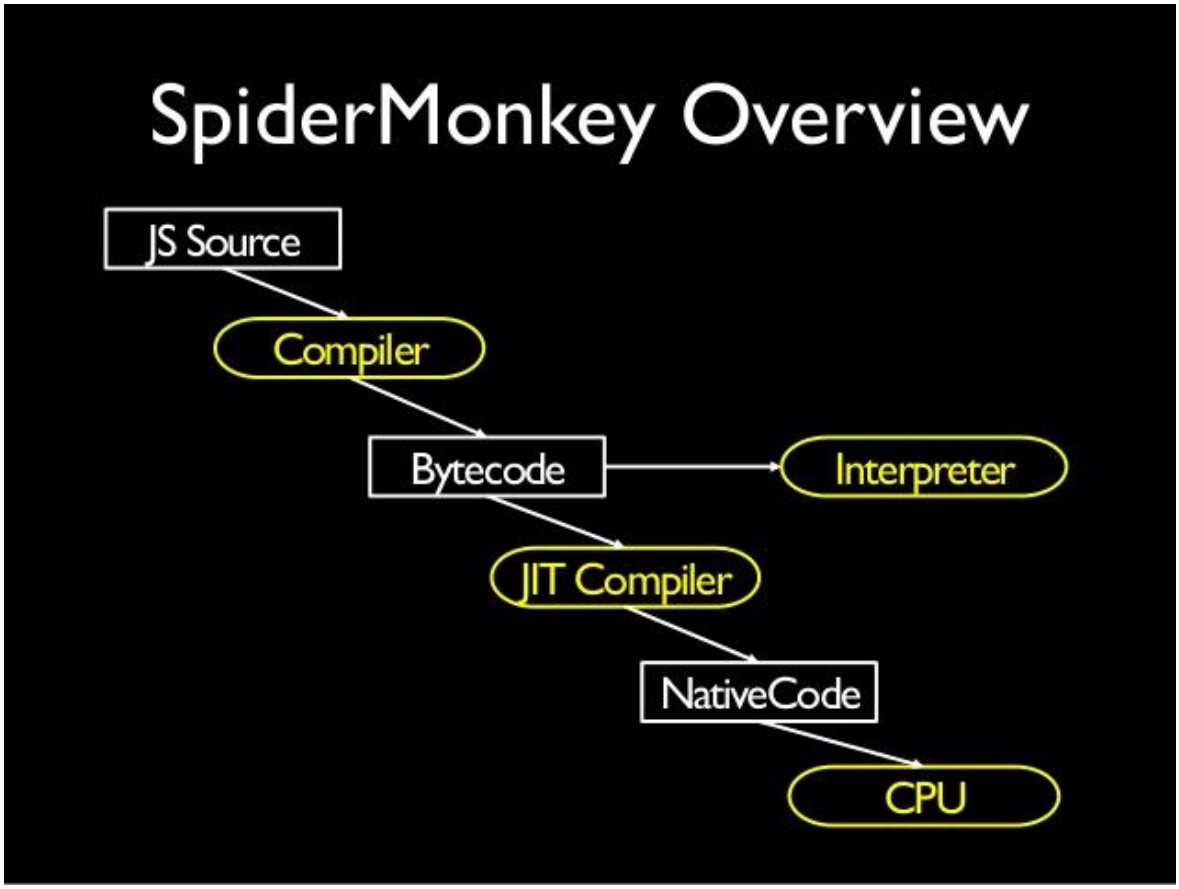

Figura 3-7: SpiderMonkey

Después de una búsqueda de más herramientas sin encontrar resultados óptimos se planteó la idea de crear nuestro propio intérprete desde cero. Parecía una idea descabellada, dado el tiempo y el resto de trabajo que quedaba por hacer no parecía muy buena idea enfrascarse en el desarrollo de algo tan grande como es un intérprete de C. Lo único que salvaba esta opción y que conseguía hacerla parecer un poco más atractiva es el hecho de que todo el código se genera a partir de nuestros bloques. Dicho de otra forma, conocemos perfectamente la estructura que va a tener el código y sabemos cómo y dónde buscar para decidir el flujo que va a seguir el programa al recorrer el código.

Finalmente, se ha optado por la opción más tediosa, pero que más se ajusta a las necesidades del proyecto. Se ha decidido crear un intérprete en JavaScript de cero, que solo contemple el formato de código que genera nuestro Blockly.

3.4 Conclusiones

Basándonos en lo comentado anteriormente, observamos cómo, aunque existen herramientas de depuración de memoria a nuestra disposición, estas son difíciles de entender y aprender a manejar dado su carácter técnico. Por ello, aprovechando el punto de apoyo que nos da el proyecto C '*Cake*' sobre la versión antigua de Blockly y la actualización de esta plataforma, hemos decidido desarrollar una versión más user-friendly para la creación de programas C y la visualización su gestión de memoria, permitiendo que aquellas personas que se estén introduciendo en el mundo de la programación tengan una base desde la que desarrollarse.

En lo referente al intérprete y a la representación gráfica se ha decidido lo que se ha decidido debido a que la generación de código se realiza con nuestras especificaciones dadas, por lo tanto sabemos cómo nos llega ese código, y cómo tenemos que devolverlo para la representación.

4 Diseño

4.1 General

4.1.1 Requisitos funcionales y no funcionales

Requisitos funcionales de la web

FN1: Permitir la navegación entre las distintas pestañas manteniendo los datos.

Requisitos funcionales de Blockly

FN2: Crear bloques que contienen significado en C.
FN3: Rellenar y modifican los bloques con los datos que se quieren aplicar.
FN4: Borran bloques o grupos de bloques.
FN5: Mover y situar los bloques en el lugar que se quiera.
FN6: Permitir la interacción con el workspace, ampliar, reducir o desplazar el mismo.
FN7: Guarda el estado actual del programa como XML.
FN8: Seleccionar un archivo XML que contenga la estructura del Blockly y cargarlo.
FN9: Habilitar o deshabilitar bloques.
FN10: Acceder al menú contextual de un bloque.

Requisitos funcionales del intérprete y la memoria

FN11: Simular la ejecución paso a paso del código mostrando la evolución de la memoria.
FN12: Modificar el espacio que ocupa cada una de las secciones de la página.
FN13: Permitir la interacción con la parte visual de cada memoria, ampliar, reducir o desplazar el contenido.

Requisitos no funcionales de la web

NFN1: Que la página sea "responsive".
NFN2: Que el uso de la web sea intuitivo y sencillo.

Requisitos no funcionales de Blockly

NFN3: Capacidad de definición de tipos propios.
NFN4: Correcto funcionamiento interno de las variables.
NFN5: Adecuar la funcionalidad de cada bloque a su contexto.
NFN6: Permitir la creación de tipos ilimitados.
NFN7: Mantener el estado de todos los bloques tras su guardado y cargado.
NFN8: Desarrollo de los bloques internamente.

Requisitos no funcionales del intérprete y la memoria

NFN9: Generar correctamente el código C a partir de los bloques creados.
NFN10: Interpretar todo el código almacenando la información a mostrar en el paso a paso.
NFN11: Resaltar la línea de código que se ejecuta en el paso a paso.
NFN12: Mostrar adecuadamente el contenido de las memorias del programa en cada paso.

4.1.2 Casos de uso

El siguiente caso de uso muestra las funcionalidades disponibles en el entorno de la web:

Figura 4-1: Caso de uso Web.

Este caso de uso se centra en las funcionalidades presentes en Blockly:

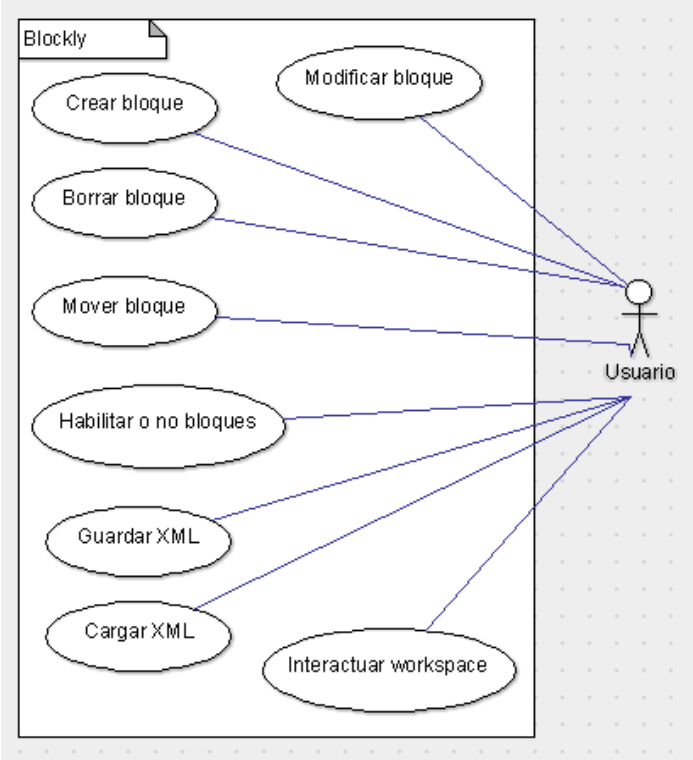

Figura 4-2: Caso de uso Blockly

El siguiente caso de uso trata de las acciones disponibles para el usuario desde la pestaña de código:

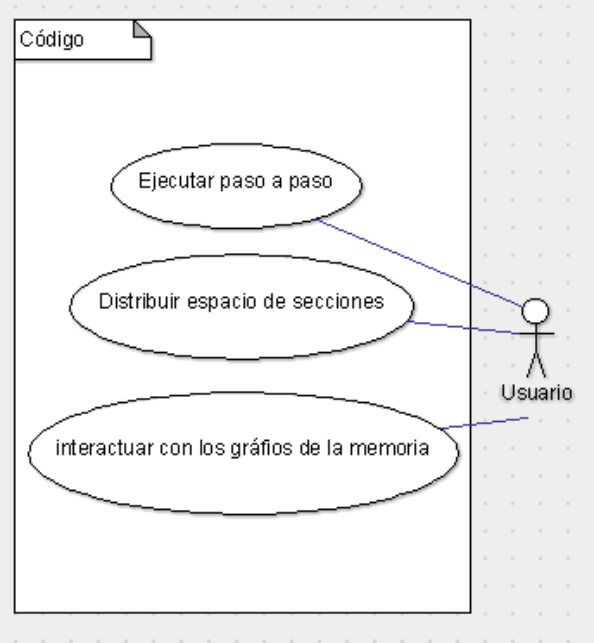

Figura 4-3: Caso de uso Código

4.1.3 Requisitos alcanzados

El diseño general de nuestra aplicación es el siguiente:

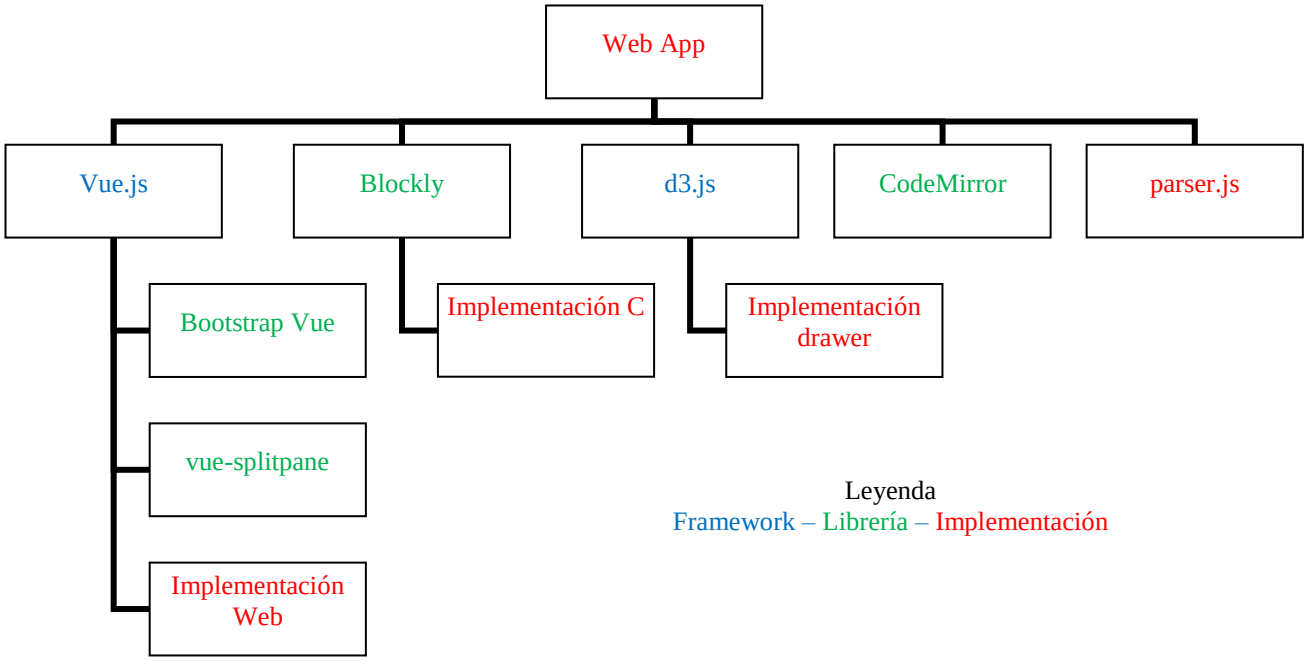

Figura 4-4: Diagrama módulos

En él podemos observar como nuestra aplicación se ha desarrollado en cuatro módulos principales:

- Mediante el framework **Vue.js**, hemos implementado el módulo web que da visibilidad al resto de ellos.
- El módulo de **Blockly**, en el cual especificamos nuestra implementación para C, así como algunos cambios personalizados sobre su librería.
- El módulo del '***parser***', en el cual analizamos el código generado por Blockly para asignarle una representación gráfica paso a paso.
- El módulo de la **representación gráfica**, donde nos encargamos de dibujar la memoria dinámica dada la salida del módulo del parser.

Dados los módulos anteriores, en la siguiente tabla se muestra dónde se ha llevado a cabo la resolución de cada requisito.

	Web	**Blockly**	**Intérprete**	**Representación gráfica**
FN1	X			
FN2		X		
FN3		X		
FN4		X		
FN5		X		
FN6		X		
FN7		X		
FN8		X		
FN9		X		
FN10		X		
FN11			X	X
FN12	X			
FN13				X
NFN1	X			
NFN2	X			
NFN3		X		
NFN4		X		
NFN5		X		
NFN6		X		
NFN7		X		
NFN8		X		
NFN9		X		
NFN10			X	
NFN11			X	X
NFN12			X	X

Tabla 4.1-1: Matriz de trazabilidad

4.1.1 Ciclo de vida del desarrollo del proyecto

Para el desarrollo del proyecto se ha seguido un ciclo de vida en espiral.

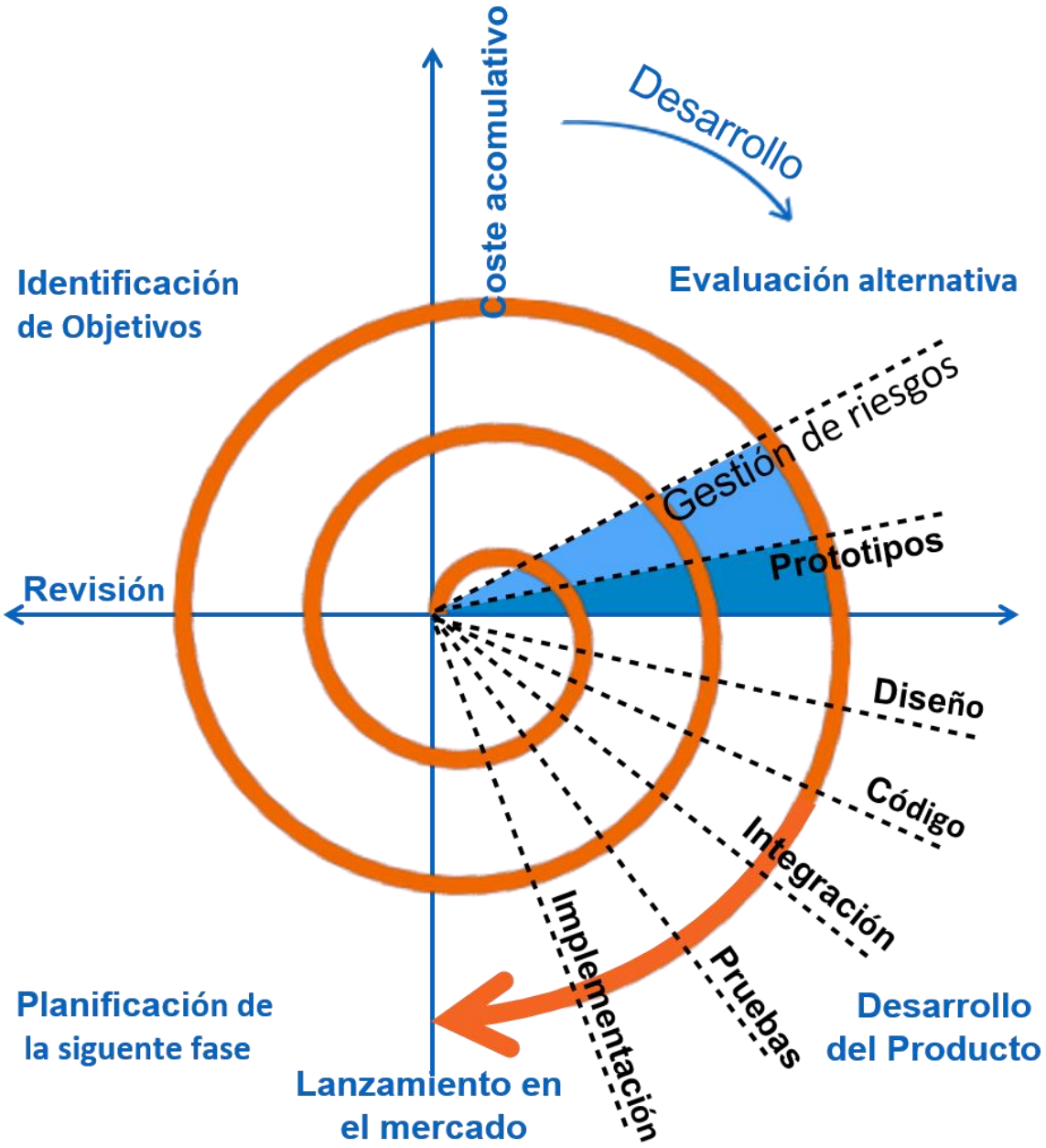

Figura 4-5: Ciclo de vida

Debido a la existencia de módulos tan distintos y que requieren de una correcta comunicación entre sí, se ha optado por un ciclo de vida basado en prototipos, para poder ir realizando pruebas conjuntas a medida que avanza el desarrollo. Y de entre las múltiples opciones se ha elegido éste por ser dinámico y flexible.

4.2 Web

4.2.1 Intérprete y representación gráfica

En la pestaña de Código podemos observar tres paneles, todos ellos redimensionables al tamaño deseado. En el primer panel (1) tenemos el código generado por Blockly, formateado de forma adecuada a sintaxis C, el cual no es modificable. En el siguiente panel (2) observamos la zona de memoria de la función main, la cual estará visible en todo momento, mientras que la zona de memoria de la función local actual (3) solo tendrá elementos si el paso actual está dentro de una función diferente al main. Las zonas de memoria tienen activado tanto la posibilidad de realizar zoom, como de moverse por el espacio.

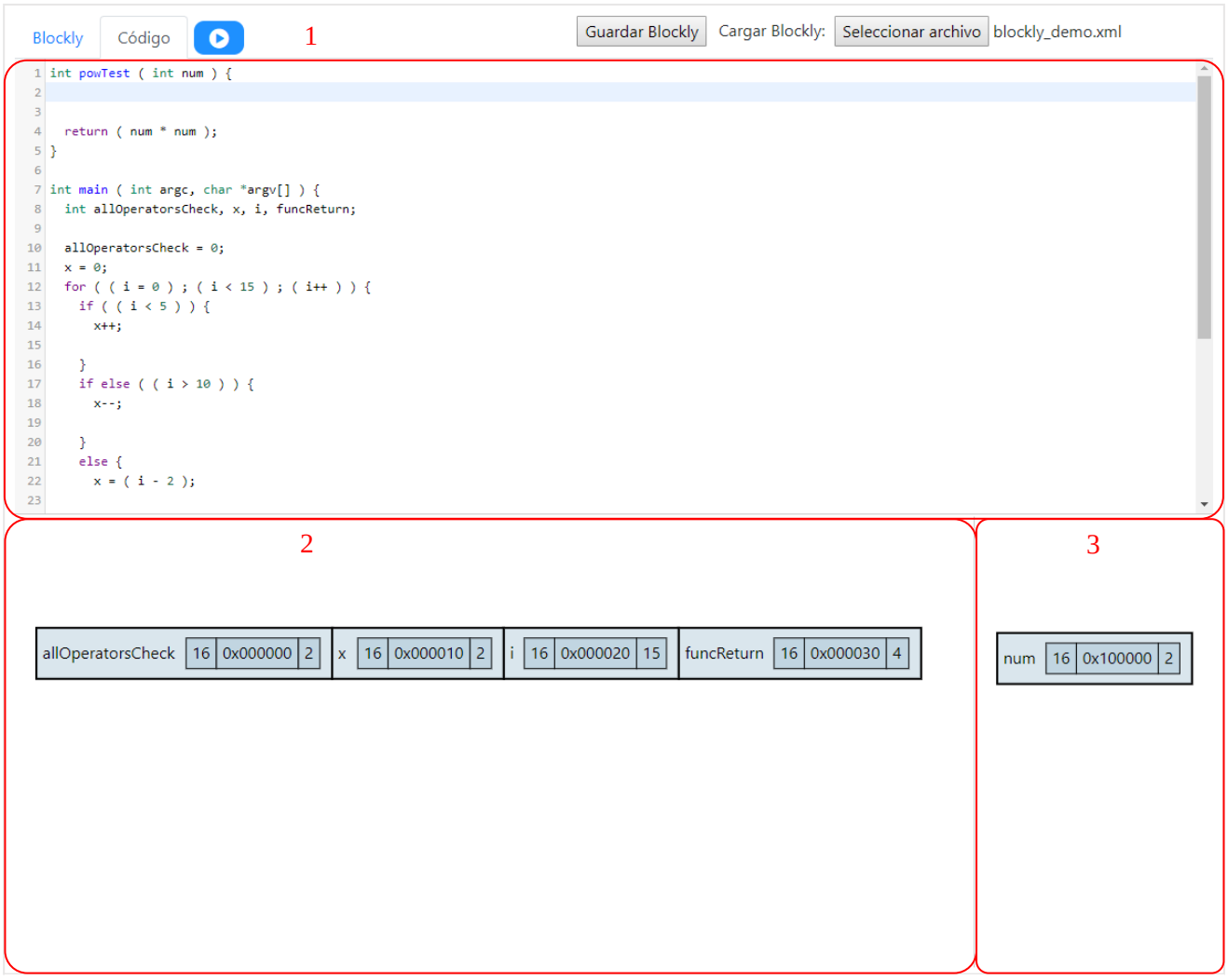

Figura 4-6: Página de código y representación gráfica

Existen diversas formas de representar una variable, todas ellas con un diseño similar. La parte común a ellas son el nombre, el tamaño que ocupan, y la dirección de memoria en la que empiezan. Después, se diferencian en variables de un solo valor, y variables con múltiples valores.

Variables de un solo valor

{ nombre, tamaño, dirección, valor }

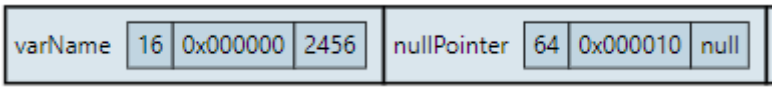

Figura 4-7: Variables de valor único

Variables de valor múltiple

{ nombre, tamaño, dirección, [valores] }

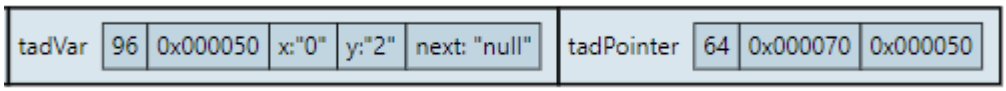

Figura 4-8: Variable de valor múltiple. TAD y puntero.

4.3 Intérprete

Se requiere de un intérprete del código generado en C que prepare los datos necesarios para representar gráficamente el estado de las memorias en cada paso de la ejecución.
En la figura 3-8 se muestra cómo debería ser la unión entre los distintos módulos que afectan al intérprete.

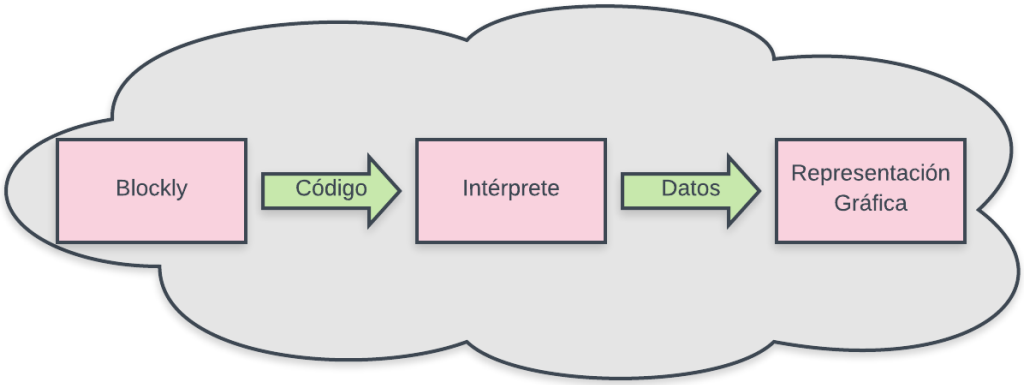

Figura 4-9: Pipeline entre módulos

Para la realización del intérprete se ha decidido crear una clase de JavaScript llamada **Programa** que contenga los atributos y los métodos necesarios para cumplir con los requisitos.
Debido a la necesidad de mantener una tabla de símbolos con los valores y los tamaños de todas las variables para pintar se ha tomado la decisión de hacer uso de la clase Map de JavaScript, esta clase permite almacenar datos de la siguiente forma {clave1 : valor1, clave2 : valor2, …} de este modo se puede acceder a los datos de forma directa con el método get, y añadir o cambiar los datos con el método set.

Otro de los requisitos es que el intérprete recorra de manera correcta el código ejecutando y evaluando todos los pasos que plantea el programa en C.
Se requiere el uso de pilas para poder llevar un control sobre los procesos que se puedan anidar o para controlar cuestiones referentes a la ejecución de las funciones. Por ello se ha decidido crear la clase *Pila*, con métodos *add*, *pop* y *getTopElement* para satisfacer esta necesidad.

Para facilitar la interpretación del programa se decide dividir la tarea en dos partes, en primer lugar, se buscan las posiciones de los elementos del código que formen bloques, y se guardan los índices de las líneas donde empiezan y donde terminan. De esta forma la segunda tarea del intérprete es más sencilla, esta tarea consiste en recorrer todo el código simulando el programa y almacenando el estado de todo en cada momento para permitir algo parecido a un "*Debugger*" a posteriori.

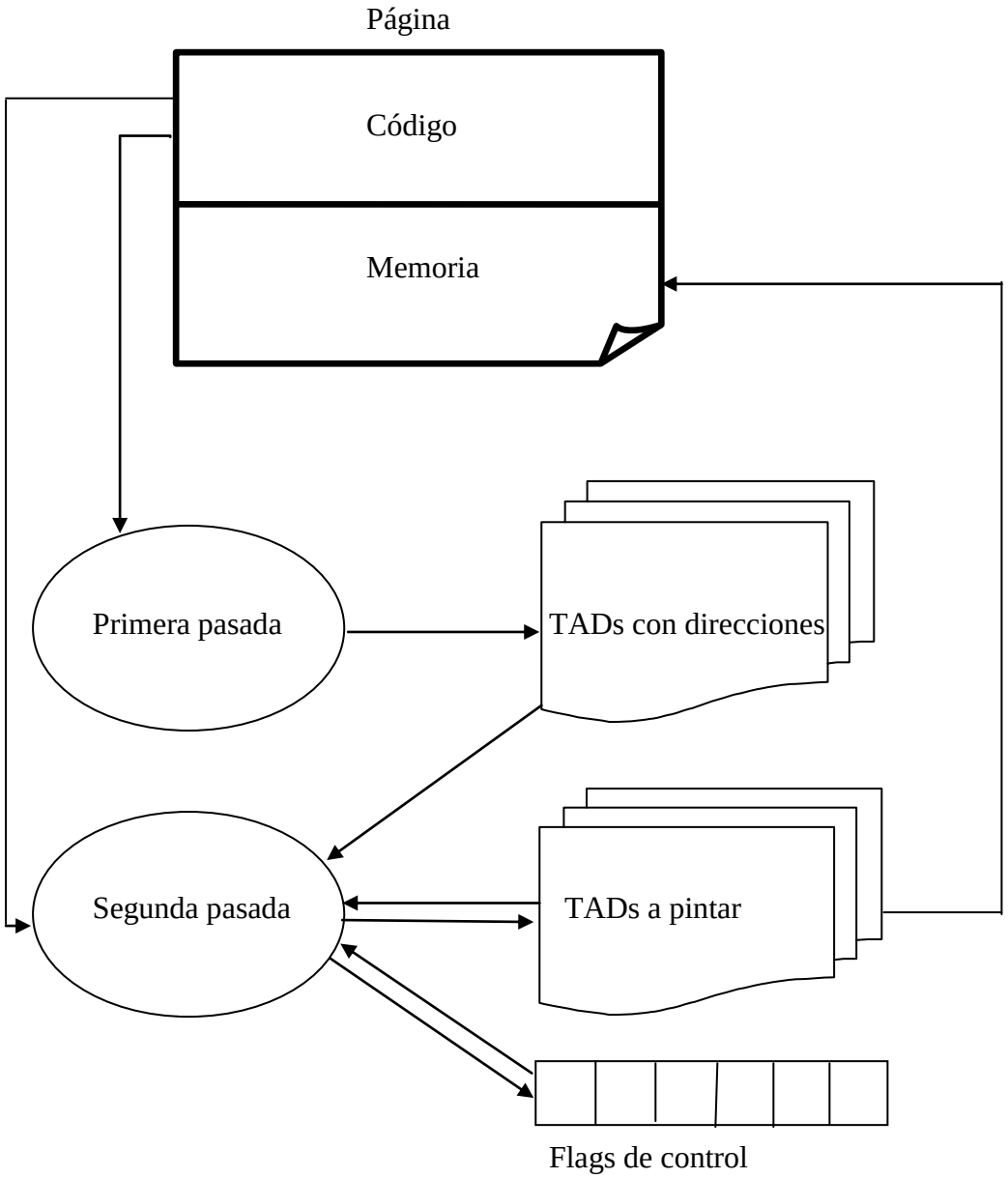

Figura 4-10: Pipeline entre módulos

La figura 3-10 muestra el flujo que sigue el intérprete y cómo es su relación con la página web que une todo el proyecto. Más adelante se detalla mejor este esquema, pero servirá para tener una idea de cómo funciona.

La ejecución sigue la siguiente estructura:

- **Bucle principal de ejecución**: Bucle que recorrerá todo el código main.
    - **Función principal**: Esta función guardará los datos necesarios para el paso a paso, y será llamada desde el bucle principal o desde procedimientos o bucles internos, define el flujo del programa.
        - **Analizar e Interpretar línea**: Será llamada por la función principal, y será la encargada de llamar a otras funciones que creen o modifiquen variables, que operen los datos. También se encarga de identificar cambios de contexto tales como saltos, bucles o llamadas a procedimientos que modifiquen el curso del programa.
    - **Función principal**: Comprueba si el paso anterior detecta algún cambio y de ser así hace las llamadas necesarias o actualiza el índice del bucle principal para controlar los saltos.
        - **Procedimiento/bucle**: Bucles que hacen llamadas a la función principal solo de sus direcciones específicas con su lógica adecuada.

Para finalizar, se ha decidido que la información de la clase Programa quede almacenada en unos Maps que tengan como clave el paso de la ejecución y como valor otro Map copia del que tenía el intérprete en el momento de ejecutar ese paso. De esta forma dejamos accesibles los datos para la parte del proyecto que se encarga de mostrar el paso a paso.

5 Desarrollo

5.1 Web

La web, como ya hemos visto, está compuesta por 2 pestañas, *Blockly* y código. Para su implementación hemos usado el *framework vue.js* [12] para el diseño gráfico de la web, el *framework d3.js* [13] para el desarrollo gráfico de la gestión de memoria, las funciones adecuadas de *Blockly* para inicializar el *workspace* y, por último, el intérprete.

5.2 Intérprete

El intérprete de *C* en *JavaScript* está situado en el fichero *parser_c.js*. En este fichero se encuentra la clase Programa, ésta contiene los atributos, los métodos, y todo lo necesario para poder representar cómo se desarrolla la reserva y gestión de la memoria del código traducido.
Las cabeceras de estos métodos se encuentran en el anexo B.

Como una de las prioridades del proyecto era la representación paso a paso de esta gestión, lo que necesitábamos era obtener en cada paso de ejecución del programa el nombre, el valor (o la dirección) y el tamaño de todas las variables que estuviesen inicializadas en ese momento, ya sea en el entorno local o en el global. Así como la posición de la línea actual de ejecución.
Para llevar a cabo esta tarea se ha decidido hacer uso de la clase *Map* de *JavaScript*, entre otras cosas.
Podemos dividir las funciones del intérprete en 3 pasos:

1) En primer lugar, se separa todo el código por líneas y se recorren todas las líneas del código almacenando las posiciones iniciales y finales tanto de los bucles como de las funciones y las estructuras de datos.

Para ello se crean los siguientes *Maps* con la siguiente estructura:

Tablas 5.2-1: Intérprete – Preparación de código

posibles_saltos	
Clave	Dirección de la línea donde se declara un salto.
Contenido	Dirección de la línea a donde se puede saltar.
Descripción	Quedan guardadas aquí todas las direcciones de los saltos *if*, *if else* y *else*.

direcciones_bucles	
Clave	Dirección de la línea donde se declara un bucle.
Contenido	Dirección de la línea a donde acaba el bucle.
Descripción	Quedan guardadas aquí todas las direcciones de los bucles for y *while*.

indice_funciones	
Clave	Nombre de la función que se declara.
Contenido	Dirección de la línea donde empieza la función.
Descripción	Quedan guardadas aquí todas las direcciones de inicio de las funciones.

fin_funciones	
Clave	Nombre de la función que se declara.
Contenido	Dirección de la línea donde acaba la función.
Descripción	Quedan guardadas aquí todas las direcciones del final de las funciones.

posicion_estructuras	
Clave	Nombre de la estructura que se declara.
Contenido	Dirección de la definición del TAD.
Descripción	Quedan guardadas aquí todas las direcciones que indican dónde se encuentra situada la definición de las estructuras en el código.

tam_estructuras	
Clave	Nombre de la estructura que se declara.
Contenido	Tamaño que ocupa toda la estructura en bits.
Descripción	Se almacena el tamaño de cada estructura para el caso de la reserva de memoria, poder acceder rápidamente al tamaño a reservar.

Estos *Maps* serán necesarios para poder realizar los saltos condicionales presentes en el programa, así como para otorgar las posiciones necesarias para llevar a cabo los bucles y las llamadas a función. Se explicará su uso dentro del intérprete más adelante.

2) En segundo lugar se recorre línea por línea, siguiendo la lógica del programa, almacenando y calculando todos los tamaños, y los valores de las variables. Todo ello se va guardando en diversos *Maps* para seguir con la interpretación del resto de la ejecución.

La información relevante de este paso queda guardada en:

Tablas 5.2-2: Intérprete – Lógica

tam_variables	
Clave	Nombre de la variable.

Contenido	Tamaño de la variable en bits.
Descripción	Con este Map se controla el tamaño que tiene cada variable en el entorno principal.

valor_variables	
Clave	Nombre de la variable.
Contenido	Valor de la variable.
Descripción	Con este Map se controla el valor que tiene cada variable en el entorno principal.

tam_variables_local	
Clave	Nombre de la variable.
Contenido	Tamaño de la variable en bits.
Descripción	Con este Map se controla el tamaño que tiene cada variable en el entorno local.

valor_variables_local	
Clave	Nombre de la variable.
Contenido	Valor de la variable.
Descripción	Con este Map se controla el valor que tiene cada variable en el entorno local.

Por cada paso de la ejecución que se va dando se van actualizando todos *Maps* anteriores, y se va guardando una copia del estado de dichos *Maps* en otros que se usarán para el paso a paso. Así como un contador total de pasos del programa, y la línea actual que se esté ejecutando para resaltarla en el código.

Los *Maps* creados en este paso son los siguientes:

Tablas 5.2-3: Intérprete – Mapas

paso_a_paso_tam_variables	
Clave	Número de paso de la ejecución.
Contenido	Copia del Map tam_variables en ese paso de ejecución.
Descripción	Con este Map se recoge el tamaño de las variables en el momento indicado de la ejecución para la representación gráfica de la memoria.

paso_a_paso_valor_variables	
Clave	Número de paso de la ejecución.

Contenido	Copia del Map valor _variables en ese paso de ejecución.
Descripción	Con este Map se recoge el valor de las variables en el momento indicado de la ejecución para la representación gráfica de la memoria.

paso_a_paso_tam_estructuras	
Clave	Número de paso de la ejecución.
Contenido	Copia del Map tam _ estructuras en ese paso de ejecución.
Descripción	Con este Map se recoge el tamaño de las estructuras en el momento indicado de la ejecución para la representación gráfica de la memoria.

paso_a_paso_tam_variables_local	
Clave	Número de paso de la ejecución.
Contenido	Copia del Map tam_variables_local en ese paso de ejecución.
Descripción	Con este Map se recoge el tamaño de las variables locales existentes en el momento indicado de la ejecución para la representación gráfica de la memoria local.

paso_a_paso_valor_variables_local	
Clave	Número de paso de la ejecución.
Contenido	Copia del Map valor_variables_local en ese paso de ejecución.
Descripción	Con este Map se recoge el valor de las variables locales existentes en el momento indicado de la ejecución para la representación gráfica de la memoria local.

paso_a_paso_linea_actual	
Clave	Número de paso de la ejecución.
Contenido	Dirección de la línea que se ejecuta en ese paso de ejecución.
Descripción	Con este Map se recoge el valor de la línea que se está ejecutando en cada paso para que pueda ser resaltada en el código C que se muestra en la pestaña.

Y esta variable:

- *contador_programa*;

Que almacena el número total de pasos que recorre el intérprete, servirá para definir el límite del paso a paso.

3) Por último, los métodos que sirven para el paso a paso, estos métodos devuelven la línea que se está ejecutando, así como los *Maps* tal y como se encuentran en el momento en que se ejecuta esa línea en el programa principal.

Tablas 5.2-4: Intérprete – Paso a paso

get_contador_programa	
Descripción	Esta función devuelve el número total de pasos que se dan en el intérprete al recorrer todo el programa. Se usa para definir el bucle exterior que sirve para pintar la memoria paso a paso.
Entrada	Ninguna.
Salida	Número total de pasos del programa.

get_linea_contador	
Descripción	Esta función devuelve la línea que se está ejecutando en el paso indicado por el contador. Sirve para resaltar la línea de código en la que se encuentra el programa durante el paso a paso.
Entrada	El paso de la ejecución programa que se quiere devolver.
Salida	La línea del código en la que se encuentra el programa en el paso determinado por la entrada.

get_tam_variables_paso, get_valor_variables_paso, get_tam_variables_local_paso, get_valor_variables_local_paso, get_tam_estructuras_paso	
Descripción	Estas funciones devuelven el contenido de los *Maps* en el paso que se está ejecutando indicado por el contador. Sirven para obtener la información necesaria para representar gráficamente cómo se desarrolla la memoria durante el trascurso del programa.
Entrada	El paso de la ejecución programa que se quiere devolver.
Salida	La copia del *Map* correspondiente.

Una vez definido qué se va a actualizar/rellenar en cada parte del código se pasa a definir de una forma más detallada el funcionamiento del intérprete de C.

Para la primera parte en la que se buscan y almacenan las posiciones de los elementos que tendrán un trato especial a la hora de interpretar, ya nombrados anteriormente (funciones, saltos, bucles y declaraciones de TADs), lo que se hace es separar el código por líneas (“\n”) y recorrerlas secuencialmente una a una.

Se buscan, en primer lugar, todos los saltos que puede haber, guardando la posición de la sentencia de salto, así como la posición a la que apunta el posible salto.
Para ello se buscan las cadenas clave: “*if*” y “*else*”; y se mantiene una cuenta de las llaves que se abren y se cierran desde cada sentencia para determinar el cierre de la misma, que supone la dirección del posible salto.

En segundo lugar, se buscan los bucles, y, al igual que con los saltos, se hace comparando con las cadenas: “*for*” y “*while*”, y manteniendo una cuenta de las llaves se obtiene la posición final del bucle. Y se guardan su posición inicial y su posición final.

Tras esto buscamos los *typedef*, comparando con la cadena "*typedef*" encontramos la declaración, y almacenamos el nombre del TAD y las variables contenidas en él (estas como una cadena). Esto es necesario para declarar cualquier variable de tipo nuestro TAD, en ese momento hay que crear cada una de las variables aquí guardadas para la nueva instancia.
También se almacena por otro lado el nombre y el tamaño de la estructura, que se usa para la reserva de memoria de los punteros.

Como último paso de este primer bloque de preparación buscamos las funciones, para ello buscamos líneas que empiecen por un tipo (o un *typedef*), que les siga un nombre y después venga un paréntesis, de cumplirse estas tres condiciones tomamos el nombre como nombre de función, y guardamos, por un lado, el nombre y la posición inicial, y por otro el nombre y la posición final, obtenida como en los casos anteriores gracias al conteo de las llaves.

Una vez que están todos los datos anteriores almacenados se procede con la simulación de la ejecución del programa.
En primer lugar, se buscan las direcciones de inicio y de final de la función "*main*", que definirán el bucle principal de todo el proceso (estas direcciones ya se encuentran en sus *Maps* correspondientes). En este bucle se llama a la función "*llamada_parser(i)*" ("i" es el índice del bucle, y es la línea del programa a evaluar), y se actualiza el valor del índice del bucle general en caso de cambio para controlar los saltos que se han podido dar durante la ejecución de este método.

La función "*llamada_parser(i)*" es la que lleva la cuenta de los pasos del programa, por eso es la encargada de rellenar los *Maps* de la parte del paso a paso y de llevar el control de la ejecución. Aquí se ignoran las líneas vacías, y los espacios y tabulados al inicio de cada una, y se llama a la función "*parser(línea, i)*", función que analiza, interpreta y evalúa cada línea del código, pero es a la vuelta de ésta cuando se comprueba si ha cambiado algo en el estado de la ejecución, la comprobación se realiza revisando el valor de una serie de *flags* (estos cambios de estado pueden ser: llamada a función, salto o bucle), y en caso de cambio llamar a los métodos que correspondan a cada estado o controlar el cambio. En el caso del salto es tan sencillo como cambiar el índice global del bucle general del programa por la dirección de la línea correspondiente al salto que toque, sin necesidad de llamar a ningún método extra. Los otros casos se desarrollarán más adelante.
La función "*parser(línea, i)*" constituye por sí sola el esqueleto del intérprete prácticamente. Consta de un "*switch case*" que direcciona cada sentencia del programa dependiendo de cómo empieza la línea. Se pueden reducir los casos a:

- **Declaración de variables**:

En este apartado se identifica el tipo de la variable declarada, y se llama a "*guardarVariables(cadena,j,tam,valor)*", en esta función comprueba si nos encontramos en un contexto local o no y si se trata de la declaración de un puntero o de un *array*, y se almacena la variable con su tamaño y su valor en sus *Maps* correspondientes. En el caso de los *arrays* y los *punteros*:

-En el caso del ***array*** el tamaño es el tamaño del tipo por el tamaño con el que se declara. Y su valor contendrá un *array*.

-En el caso del ***puntero*** el tamaño será el del tipo de este. Y su valor será a lo que apunte, en la declaración *null*.

- **Saltos**:

En este caso se identifica el tipo de salto ("*if*", "*if else*", "*else*"), y se calcula la condición de salto llamando a "*operaciones(cadena_entrada)*" (el funcionamiento de la función "*operaciones(cadena_entrada)*" se explica más adelante) en caso del "*if*", comprobando además los saltos precedentes en caso del "*if else*", o solamente esto último en caso del "*else*". Con ese fin se almacenan en una pila el resultado de las condiciones de los "*if*" y los "*if else*" para poder calcular la lógica de los próximos saltos.

Para marcar que el salto se cumple se activa un *flag* "*salto*", y se guarda la salida de la condición en una pila de antiguos saltos para poder calcular los próximos que dependan de ellos. La función "*llamada_parser(i)*" es la encargada de recoger ese *flag* activo y recoger la dirección de la próxima línea a interpretar para actualizar el flujo de la ejecución. Cuando se ha actualizado la posición del programa se vuelve a desactivar el *flag* "*salto*".

También se almacena en una pila de llamadas ("*pila_llamadas*") el comando en que se encuentra, en este caso las opciones son "*if*", "*ifelse*" y "*else*". La información de esta pila será fundamental tanto para conocer en qué estado se encuentra la ejecución, tanto como para los cierres de los bloques de código. Para ello también se mantiene una cuenta de las llaves que se van abriendo de cada caso.

- **Bucles**:

Los dos bucles que se contemplan en el proyecto se tratan de manera muy distinta, por tanto, se explican por separado.

-**While**:

Tras identificar el caso se procede a actualizar los contadores oportunos de llaves y de bucles, y se guarda el tipo de llamada en su pila determinada.
Se prepara la condición del bucle en un *Map* con la dirección de la línea como clave, y se añade esta dirección a la pila de bucles ("*pilaBucles*").

Por último, la función "*parser(línea, i)*" activa el *flag* "*bucleWhile*" y deja que el método que ha realizado la llamada ("*llamada_parser(i)*") se ocupe de identificar el *flag* activo, desactivarlo por si hubiese nuevos bucles, y llamar al método que procesa este tipo de bucle. Este método es "*bucleWhileFn()*" (esta función se explica más adelante).

-For:

De igual forma que en el caso anterior, lo primero que se hace cuando se reconoce una sentencia "*for*" es, actualizar los contadores oportunos, y apilar el tipo de llamada.
Posteriormente se guardan por separado las tres partes del bucle, a saber, la inicialización del índice, la condición de parada, y la acción que se realiza al finalizar cada iteración del bucle.

Después, igual que en el caso anterior se inserta en la pila de bucles la dirección de la línea, y se activa el *flag* "*bucleFor*".
Finalmente, es otra vez "*llamada_parser(i)*" el método encargado de detectar éste *flag*, desactivarlo por si hubiese nuevos bucles, y llamar a las funciones necesarias,

en este caso son dos: "*prepararFor(i)*" y "*bucleForFn( )*" (estas funciones también se explicarán después).

- **Cierre de bloque**:

La función "*parser(línea,i)*" entra en este caso cuando encuentra un símbolo de cierre de llave "}". Extrae de la pila de llamadas el bloque que se está cerrando y con un "*switch case*" se divide en los distintos casos:

-**Estructura**:
Se usa para indicar que la declaración de variables de dentro de una estructura acaba. Para ello se desactiva el *flag* correspondiente y se actualiza el número de llaves.
-**Función**:
En este caso se saca de la pila la función, se actualiza el número de llaves, así como los contadores de funciones anidadas, y se eliminan de la memoria local las variables declaradas en el entorno. También se comprueba si se tiene que actualizar el *flag* "*isLocalVariable*" que determina si estamos en el entorno local o no dependiendo de si el contador de funciones está a 0.
-**If**:
Se actualizan los contadores de llaves, y de "*if*", y se extrae de la pila "*ultimo_if*" el resultado de la condición para posibles "*if else*" o "*else*" tras él y se guarda el resultado en la variable "*ultimo_salto*".
-**If else**:
Se actualizan los contadores de llaves, y de "*ifelse*", y como en el caso anterior se extrae de la pila "*ultimo_if*" la salida de este salto, pero solo se actualiza con ese resultado la variable "*ultimo_salto*" en caso de que esta última no valiese ya "*true*".
-**Else**:
Simplemente se actualizan los contadores y los *flags* correspondientes.
-**For**:
Se actualizan los contadores de llaves y de bucles, y se extrae de la pila la dirección de éste.
-**While**:
Igual que en el caso anterior, se actualizan los contadores y la pila.
-**Main**:
En este caso solo se controla el contador de llaves y se acaba la ejecución.

- **Free**:

Cuando se detecta un "*free*" la función "*parser(línea,i)*" identifica si el puntero está en memoria local o no, vacía el contenido al que apunta y pasa a apuntar a *null*.

- **Resto de sentencias**:

Por este caso entran: el resto de las declaraciones, las asignaciones y las llamadas a función.
Se determina en cuál de los tres casos se encuentra buscando si la primera palabra de la línea es respectivamente en cada caso: el nombre de una estructura, el nombre de una variable (local o no local), o el nombre de una función.

-**Resto de declaraciones**:
Quedan por declarar las variables de tipo TAD que no entran en la condición anterior, para ello se recoge la dirección del "*typedef*", se guarda el nombre de la variable a guardar en "*nombre_tad*", se activa el *flag* "*declaracion_tad*" y se llama

a "*parser(línea,i)*" mientras el *flag* "*declaracion_tad*" siga activo con las líneas que siguen a la dirección recogida del *typedef*.
En estas llamadas a '*parser*' la declaración sí que entra en el caso de declaración anterior, y se realiza la llamada a "*guardarVariables(cadena,j,tam,valor)*". Esta función tiene el mismo funcionamiento de antes, pero cuando el *flag* "*declaración_tad*" está activo se crean las variables concatenando el nombre guardado en "*nombre_tad*" con el nombre de cada variable interna con un punto (Por ejemplo "*p.x*" y "*p.y*", serían las variables que componen "*p*", que es de tipo "*punto*", un TAD), también se guarda el tamaño de la estructura en conjunto, es decir, el tamaño de una variable de tipo una estructura de datos será la suma de los tamaños de todas las variables que la formen.

-**Asignaciones**:
En las asignaciones lo primero que comprobamos es si se trata de un "*malloc*", en cuyo caso llamamos a la función "*reserva_memoria(size)*" que calculará el tamaño a reservar, y llamará a "*guardarVariables(cadena,j,tam,valor)*" con el nombre de variable el nombre del puntero que lo apunta precedido de "*" y seguido de "[", tamaño del *array* a generar y "]", es decir, si hacemos un "*malloc*" a la variable "*puntero*", en verdad lo que se crea es un *array* de nombre "*puntero", y tamaño el indicado. Y se guarda en el valor del puntero el nombre de la variable nueva creada.

El segundo caso que se comprueba es si se trata de una llamada a función con retorno. En ese caso se almacena en nombre de la variable destino de la asignación en "*pilaVariablesRetorno*", y se apila un "true" en "*pilaHayRetorno*".
Y tras eso se llama al método "*operaciones(cadena_entrada)*" en este método se identifica si solo está la función, o si hay más operadores, y se realiza lo necesario para las llamadas a funciones (todo ello se explica más adelante), también realiza la lógica necesaria para el resto de las asignaciones.

-**Llamadas a función sin retorno**:
Como no hay retorno se apila un "false" en "*pilaHayRetorno*".
Se realiza una llamada a "*llamada_funcion(i)*", en esta función se cuentan y guardan los parámetros de llamada, se activan los *flags* "*llamada_a_funcion*" e "*isLocalVariable*", y se incluye la llamada a función en la "*pila_llamadas*", también se apila el nombre de la función en "*pilaNombre_funcion*" y se actualiza el contador de funciones.
Tras esto, cuando el programa regresa a "*llamada_parser(i)*" y el *flag* "*llamada_a_funcion*" está activado se realiza la llamada a "*función(inicio,final)*" (el funcionamiento de este método se explicará después).

Cuando la ejecución del intérprete sale de la función "*parser(línea,i)*" y regresa a "*llamada_parser(i)*" es el momento en el que se evalúan los *flags* que determinan los sucesos del programa, como ya hemos visto estos casos pueden ser: saltos, bucle *while*, bucle *for* y *funciones*.

- **Saltos:**

En el caso de los saltos ya se ha dicho que sirve con recoger la dirección de salto y volver a desactivar el *flag* correspondiente.

- **Bucle while**:

En este caso se desactiva el *flag* correspondiente y se llama a la función "*bucleWhileFn()*" que devuelve la posición en la que acaba el bucle.
Esta función recoge por un lado la condición guardada anteriormente, y por otro la dirección de la última línea del bucle.
La condición pasa por "*operaciones(cadena_entrada)*" y lo que devuelve es la condición de un bucle *while* que incluye un bucle *for*, que recorre las líneas del bucle llamando a "*llamada_parser(i)*".

- **Bucle for:**

Lo primero se desactiva el *flag*, después se llama a la función "*prepararFor(i)*" que se encarga de extraer y guardar los datos necesarios para la ejecución del bucle en una serie de *Maps* con clave la dirección del bucle.
-En "*mapVariableBucleFor*" se guarda el nombre de la variable que hace de índice.
-En "*mapComparadorBucleFor*" se guarda el comparador.
-En "*mapLimiteBucleFor*" se guarda el valor con el que se compara.
-Y en "*mapAccionFinalBucle*" se guarda la acción que se realiza al final de cada iteración del bucle.

Una vez guardados estos datos la función "*llamada_parser(i)*" llama a "*bucleForFn()*" en esta función se recoge la dirección final del bucle y se genera un bucle *for* con los datos preparados anteriormente, y otro dentro anidado que recorre las líneas de código que forman el bucle haciendo llamadas a "*llamada_parser(i)*".

- **Funciones:**

En primer lugar se desactiva el *flag* de "*llamada_a_funcion*", se extrae de la pila "*pilaHayRetorno*" la información de si hay retorno o no, en caso afirmativo se extrae dela pila "*pilaVariablesRetorno*" el nombre de la variable y se realiza la llamada a la función "*función(inicio,final)*" asignando su retorno a la variable controlando si nos encontramos en el entorno local o no.

En caso de no haber retorno simplemente se llama a la función. Para las llamadas a esta función se recogen su inicio y su final de los "*Maps*" creados al principio.
En este método lo primero que se hace es recoger los parámetros y crear las variables locales necesarias con los valores recogidos de las pilas de parámetros haciendo llamadas a "*guardarVariables(cadena,j,tam,valor)*".

Una vez preparados los parámetros se recorre un bucle desde la posición inicial de la función hasta la final realizando llamadas a "*llamada_parser(i)*" y aquí aparece otra de las funciones de llamada, si "*llamada_parser(i)*" encuentra algún "*return*" este método lo calcula y devuelve algo distinto de "-1", en caso contrario no hay "*return*".

Queda por explicar el método "*operaciones(cadena_entrada)*", este método agrupa operaciones por paréntesis, y va realizando llamadas recursivas a sí mismo hasta que llega al caso en que solo hay uno o dos operadores. Cuando solo queda un elemento lo evalúa (puede tratarse de una función, o de una asignación de puntero, "&" o "*"), y devuelve el valor. En caso de tratarse de dos recoge ambos por separado, busca a ver si son variables, locales o no, o literales, así como la operación y llama a "*operar(op1,op,op2)"* que devuelve el resultado. De esta forma se cubren todas las operaciones contempladas en nuestro proyecto y el control sobre la memoria dinámica a tratar.

Ahora que ya se conoce un poco más el funcionamiento interno del intérprete, podemos intentar imaginar desde fuera el flujo que mantendría el mismo a la hora de procesar un código C cualquiera.
Una vez creada la clase Programa, clase que contiene todo lo nombrado en esta sección, y que se ha llamado a su constructor con el código generado por Blockly ya tenemos todos los datos necesarios para la representación gráfica de la memoria en cada paso. Esta clase contiene los métodos a los que llamará la aplicación Web para la construcción de las tablas que representan la memoria, y cuyo desarrollo se cuenta en otra sección.

5.3 representación gráfica

En la pestaña de '*Código*' podemos ver el código generado, así como representaciones de la memoria del *main* y local de la función actual. Conseguimos que el tamaño de cada zona sea variable usando *vue-splitpane* [18], lo que nos permite fácilmente configurar los paneles mediante las etiquetas HTML <*split-pane*> y <*template*>.

La salida del código viene dada por *Blockly*, y usamos el editor de texto en JavaScript *CodeMirror* para formatear el código de la manera adecuada, usando el modo de lenguaje de C [19]. También es usado este editor para marcar la línea activa en el intérprete cuando se pulsa el botón de siguiente paso, el cual llama a un *callback* situado en la aplicación web principal. Por otro lado, las representaciones gráficas de la memoria del main y la local viene dada por el módulo *Draw*, desarrollado en el fichero '*main_d3.js*', el cual pone a nuestra disposición una interfaz mediante la cual podemos dibujar en el *canvas* programáticamente. La interfaz está compuesta por las siguientes funciones:

Draw.init( )

Esta función se encarga de inicializar todos los parámetros necesarios, así como de realizar las inicializaciones de los eventos necesarios para realizar el movimiento y zoom de la zona de dibujo. Esta función debe ser llamada de manera única cuando se inicialice la pestaña.

Draw.clean( )

Esta función actúa de manera similar a *Draw.init( )*, con la diferencia de que solo reinicializa los datos necesarios para vaciar el tablero, manteniendo todos los eventos declarados con anterioridad. Esta función puede ser llamada tantas veces como se desee.

Draw.variable(varName, varSize, varValue, zone)

Esta función se encarga de pintar una variable en la zona de dibujo. La nueva figura se pintará automáticamente en la siguiente posición libre, siguiendo el límite de ancho en el eje X impuesto en la constante *Draw.LIMITX*.

Las variables necesarias en esta función son*:*

- ***varName***: Nombre de la variable.
- ***varSize***: Tamaño de la variable.
- ***varValue***: Valor de la variable.
- ***zone*** *(opcional)*: Valor por defecto: *Draw.GLOBAL*. Indica en qué zona pintar la variable, si en la global correspondiente al *main* (*Draw.GLOBAL*), o la zona local correspondiente a la función ejecutándose actualmente (*Draw.LOCAL*). Hemos tomado como referencia de partida *0x000000* para la memoria global, y *0x100000* para la local.

Draw.variableMultiple(varName, varSize, numValues, varValues, zone)

De manera similar a la función anterior, esta función se encarga de pintar una variable en la zona de dibujo. Sin embargo, a diferencia del caso anterior, se le pasa como parámetro una serie de valores que representar.

Las variables necesarias en esta función son:

- ***varName***: Nombre de la variable.
- ***varSize***: Tamaño de cada valor de la variable.
- ***numValues***: Número de variables presentes en el *array*.
- ***varValues***: *Array* de valores de la variable.
- ***zone*** *(opcional)*: Valor por defecto: *Draw.GLOBAL*. Indica en qué zona pintar la variable, si en la global correspondiente al *main* (*Draw.GLOBAL*), o la zona local correspondiente a la función ejecutándose actualmente (*Draw.LOCAL*).

Así mismo, tenemos a nuestra disposición una serie de constantes y variables que podemos modificar para alterar el funcionamiento de estas funciones:

- ***Draw.LIMITX*** *(default: 2000)*: Esta constante indica el límite de anchura que alcanzará cada línea de bloques en las zonas de dibujo.
- ***Draw.VARHEIGHT*** *(default: 50)*: Esta constante indica la altura que tendrá cada bloque de variable.
- ***Draw.variableBackground*** *(default: "#dae6ed")*: Esta variable indica, en forma de *string*, el código de color RGB que se quiera poner en la variable.
- ***Draw.memBackground*** *(default: "#c1d5e1")*: Esta variable indica, en forma de *string*, el código de color RGB que se quiera poner dentro de la variable.
- ***Draw.borderColor*** *(default: "# 000000")*: Esta variable indica, en forma de *string*, el código de color RGB que se quiera poner en el contorno de la variable.
- ***Draw.borderWidth*** *(default: 2)*: Indica el número de píxeles de anchura del borde exterior de la variable.

6 Integración, pruebas y resultados

6.1 Integración

Para empezar el desarrollo de la aplicación web, empezamos a desarrollar los módulos de manera paralela, haciendo en cada uno de ellos las pruebas correspondientes y usando una web provisional. Una vez todos los módulos estuvieron en un estado de maduración suficiente, desarrollamos la plataforma final donde ubicar cada uno de ellos usando *vue.js* y comprobando el correcto cambio entre pestañas sin contenido. Una vez lo tuvimos terminado, rellenamos los huecos con los módulos, y durante este proceso nos encontramos con algunos problemas de visualización al cambiar de pestañas, así como entre el código generado por Blockly con el intérprete. Antes de continuar con el desarrollo nos centramos en conseguir una integración completa entre los diferentes módulos, haciendo en el proceso una demo lo más completa posible.

6.2 Pruebas y resultados

.

6.2.1 Representación gráfica

Para probar la correcta generación de los gráficos de memoria, creamos una función interna, '*Draw.drawing_'* que se encarga de generar una serie de variables, de forma que use todos los tipos de variable y pase por todas las líneas de su código, comprobando su correcta visualización, tanto en forma como en movilidad. También se han realizado pruebas individuales, añadiendo de forma manual distintas variables mediante la consola del navegador. La utilización de valores inválidos en los parámetros de creación de variables no genera errores. Sin embargo, esto puede llevar a comportamientos anómalos. Mediante las pruebas hemos detectado un bug exclusivo de la zona local de memoria, el cual provoca que, si su anchura es demasiado pequeña al ser redimensionado su panel, parte del contenido de cada variable se vuelve invisible.

6.2.2 Intérprete

A medida que el desarrollo de esta sección ha avanzado se han ido realizando pruebas básicas de funcionamiento de cada método que se ha ido completando.
En varias ocasiones se ha vuelto a reabrir el desarrollo de algún método ya acabado y probado para añadir alguna funcionalidad extra y se han vuelto a realizar pruebas de cada acción que se contemplase en el procedimiento.

Cuando estaban finalizados varios módulos que se combinaban uno con otro se han realizado pruebas de ambos en conjunto.

Estos dos casos anteriores han marcado el flujo de trabajo de esta sección hasta que se ha tenido todo acabado y se han podido realizar las pruebas en conjunto. Estas pruebas han sido muy exhaustivas, probando casi la totalidad de las opciones que da el código. En estos casos de prueba se han detectado muchos errores y necesidades que ha habido resolver e incluir, para volver a probar por separado y en conjunto hasta que se ha acabado satisfecho con el resultado.

6.2.3 Común

Tras estas pruebas hemos pasado a las de integración con el resto del proyecto, para esto se ha generado una plantilla de *blockly* que contempla todos los casos de ejecución y se ha generado el código y pasado al intérprete. En estas pruebas se han encontrado errores de compatibilidad triviales entre lo generado en un módulo y lo esperado en el otro, se han resuelto estos fallos y se han vuelto a realizar las pruebas hasta acabar conformes.

Cuando todo ha estado probado y hemos aprobado el producto se han comenzado con las pruebas de aceptación, se ha buscado gente de diferentes ámbitos profesionales y se les ha pedido que prueben el producto. La gran mayoría ha aprendido a usarlo rápidamente y han mostrado su conformidad con el estado final del mismo.

7 Conclusiones y trabajo futuro

7.1 Conclusiones

Las aplicaciones web son un método útil y ágil de trabajo que nos permiten desarrollar casi cualquier cosa. El uso de *Frameworks* y de librerías externas facilitan el desarrollo de múltiples herramientas para lograr fines más o menos complejos.

En nuestro caso haber desarrollado una aplicación educativa en el entorno web ofrece la posibilidad de que cualquier usuario con acceso a internet mediante un navegador pueda hacer uso de la misma y beneficiarse de las herramientas que hemos desarrollado y enlazado. Estas herramientas proporcionan una ayuda de especial interés a la hora de aprender y afianzar los conceptos básicos de la programación secuencial y con punteros. Permite al usuario un modo de crear código complejo y bien formado sin necesidad de conocimientos avanzados del lenguaje y observar la evolución del programa gracias a la visualización paso a paso de la línea en ejecución y del estado de las memorias del programa. De este modo el usuario será consciente del funcionamiento del lenguaje y será capaz de crear sus propios programas.

El uso de *blockly* para generar el código mediante cajas ha supuesto una ayuda a nuestro desarrollo, pero también nos ha obligado a indagar acerca de su funcionamiento interno para cambiar cosas como los tipos y las variables. Debido a la poca documentación proporcionada por los desarrolladores de la herramienta se ha requerido un proceso de ingeniería inversa para identificar los cambios necesarios para que sirviese con un lenguaje tipado como C.

Por otro lado, la necesidad de obtener la información correspondiente a cada variable y al estado de ejecución en cada paso supuso una búsqueda infructuosa de herramientas en JavaScript que solventasen ese problema. Por ese motivo se decidió crear un intérprete propio que con la entrada de nuestro código generado calculase todo para poder devolver los datos requeridos en cada momento. Esto supuso una tarea extra de análisis y diseño para poder empezar con el desarrollo del mismo. Se ha echado en falta la existencia de una herramienta cumpliese con estas características.

Se ha buscado crear una herramienta para programación II de grado en ingeniería informática.

Por último, creemos que el proyecto tiene grandes posibilidades para afianzarse como tutorial básico de los principios de la programación en lenguaje C, uso de punteros y gestión de la memoria dinámica. El proyecto es ampliable tanto a otros lenguajes como a la integración de librerías externas para aumentar su funcionalidad.

7.2 Trabajo futuro

Como opciones para trabajo en el futuro se plantea, realizar los cambios necesarios en *Blockly* para poder trabajar con distintos módulos a la vez. Mejorar los bloques de *Blockly* para controlar dinámicamente los tipos, así como añadir más dinamismo y avisar de un mayor número de errores lógicos en los bloques al usuario.

Incluir más opciones de depuración como por ejemplo permitir entrar o no en los métodos, o retroceder en la ejecución. La posibilidad de avanzar hasta el final directamente o hasta algún "*breakpoint*". Esto facilitaría el uso de la herramienta, ya que ahora mismo si tenemos un programa grande tenemos que recorrer la ejecución paso a paso entrando en todas las sentencias del código las veces necesarias hasta llegar a donde nos interese.

También podría estar bien la opción de cambiar algún valor directamente en el código o en la memoria para realizar mejor las pruebas, no ha sido un requisito para nuestro proyecto, pero sería interesante permitir el cambio de los valores en medio de la ejecución. Esto permitiría, junto con la posibilidad de avanzar hacia atrás y hacia adelante en el paso a paso, la posibilidad de mejorar el proceso de desarrollo de un algoritmo, ya que serían herramientas muy útiles y sencillas.

La posibilidad de ampliar este proyecto a otros lenguajes, de admitir la inclusión de librerías externas y de añadir lo comentado anteriormente podría dar pie a la presentación de nuevas opciones de proyectos futuros.

El software del producto se ha desarrollado pensando en el soporte y en la escalabilidad, se ha modularizado y comentado con el fin de facilitar los futuros desarrollos sobre el mismo.

Debido a la ausencia de un proyecto similar en código abierto, podría ser interesante abrirlo a la comunidad, como pudiese ser en "*GitHub*", para dar un empuje a su desarrollo y facilitar su mantenimiento, todo ello tras un proceso de documentación y conversión al inglés.

Referencias

[1] Google, «Blockly | Google Developers» [En línea]. Disponible: https://developers.google.com/blockly/.

[2] Massachusetts Institute of Technology, «MIT App Inventor» [En línea]. Disponible: http://appinventor.mit.edu/explore/index-2.html.

[3] Google, «Blockly Demo: Code» [En línea]. Disponible: https://blockly-demo.appspot.com/static/demos/code/index.html

[4] Computer Research Association, «cake-core» [GitHub]. Disponible: https://github.com/cra16/cake-core

[5] Valgrind, «Valgrind» [En línea]. Disponible: http://valgrind.org/

[6] Josef Weidendorfer, «KCachegrind» [En línea]. Disponible: https://kcachegrind.github.io/html/Home.html

[7] Free Software Foundation, «DDD – Data Display Debugger - GNU Project - Free Software Foundation» [En línea]. Disponible: https://www.gnu.org/software/ddd/

[8] Free Software Foundation, «GDB: The GNU Project Debugger» [En línea]. Disponible: https://www.gnu.org/software/gdb/

[9] SoftIntegration, «Ch -- an embeddable C/C++ interpreter, C and C++ scripting language» [En línea]. Disponible: http://www.softintegration.com/

[10] Fabrice Bellard, «TCC: Tiny C Compiler» [En línea]. Disponible: https://bellard.org/tcc/

[11] Google, «Toolbox | Blockly | Google Developers» [En línea]. Disponible: https://developers.google.com/blockly/guides/configure/web/toolbox

[12] Evan You, «Vue.js» [En línea]. Disponible: https://vuejs.org/

[13] Mike Bostock, «D3.js - Data-Drivent Documents» [En línea]. Disponible: https://d3js.org/

[14] BootstrapVue, «Bootstrap Vue» [En línea]. Disponible: https://bootstrap-vue.js.org/

[15] Kevin David, «Modelos de ciclo de vida del Software» [En línea]. Disponible: https://www.mindomo.com/es/mindmap/fase-comunes-entre-mocelos-de-ciclo-de-vida-del-software-f29d5494c5d146d5a5636f7b3e4a9091

[16] Zaach, «Bison in JavaScript» [En línea]. Disponible: https://github.com/zaach/jison

[17] Ricardoquesada, «SpiderMonkey» [En línea]. Disponible: https://github.com/ricardoquesada/Spidermonkey

[18] PanJiaChen, «vue-splitpane» [GitHub]. Disponible: https://github.com/PanJiaChen/vue-split-pane

[19] Marijn Haverbeke, «CodeMirror: C-like mode» [En línea]. Disponible: https://codemirror.net/mode/clike/

[20] Iván Serrano Sagredo «Aplicación web para ayuda en el aprendizaje de la gestión de memoria dinámica en programación con el lenguaje C» Depositado en la secretaría del departamento de Ingeniaría Informática de la Escuela Politécnica Superior de la Universidad Autónoma de Madrid.

Glosario

API	Application Programming Interface
DDD	Data Display Debugger
GDB	GNU Debugger
MIT	Massachusetts Institute of Technology
HTML	HyperText Markup Language
XML	Extensible Markup Language
DOM	Document Object Model
TAD	Tipo Abstracto de Datos

Anexos

A Manual de uso

Se encuentra en la memoria de Ivan Serrano Sagredo.

B Funciones del intérprete

El intérprete, como ya se ha explicado, consta de la clase Programa, y estos son sus métodos:

Tablas B-1: Funciones del intérprete

constructor	
Descripción	Este es el método principal de la clase, recibe el código, prepara las direcciones de los saltos, y lanza el bucle principal de la ejecución.
Inputs	Cadena con todo el código en C.
Outputs	Sin salida.

funcion	
Descripción	Este método simula la llamada a función. Recoge las posiciones inicial y final, almacena los parámetros y recorre las líneas que le corresponden.
Inputs	Inicio: posición inicial de la función Final: posición final de la función.
Outputs	El retorno de la función.

bucleForFn	
Descripción	Este método simula la llamada la ejecución de un bucle for en C a partir de un bucle en JavaScript. Y recorre las líneas que le corresponden.
Inputs	Ninguno.
Outputs	Posición final del bucle.

bucleWhileFn	
Descripción	Este método simula la llamada la ejecución de un bucle while en C a partir de un bucle en JavaScript. Y recorre las líneas que le corresponden.
Inputs	Ninguno.
Outputs	Posición final del bucle.

prepararFor	
Descripción	Este método prepara los datos para la llamada a la función que recorre el for.
Inputs	Posición de llamada al bucle for.
Outputs	Ninguno.

isTipe	
Descripción	Método que comprueba si una cadena es un tipo de C o un TAD definido.
Inputs	Cadena a comparar.
Outputs	True: si es un tipo. False: si no es un tipo.

guardarVariables	
Descripción	Método que guarda una variable o un grupo de variables del mismo tipo comprobando si es de una estructura, si es local o no o si se trata de un puntero o un array.
Inputs	Cadena a guardar. Posición en la cadena. Tamaño del tipo que se guarda. Valor de la variable a guardar.
Outputs	Ninguno.

operaciones	
Descripción	Método que interpreta las cadenas de operaciones, las llamadas a funciones y las asignaciones de punteros o de arrays.
Inputs	Cadena a evaluar.
Outputs	Salida de la operación o valor a asignar.

operar	
Descripción	Método que realiza las operaciones básicas.
Inputs	Operando 1. Operador. Operando 2.
Outputs	Salida de la operación.

llamada_parser	
Descripción	Método que almacena los datos para el paso a paso y comprueba si hay saltos, bucles o llamadas a función tras el análisis de cada línea.
Inputs	Índice de la línea.
Outputs	-1 si nada Valor retorno si se trata de un retorno de función.

parser	
Descripción	Método que analiza cada línea identificando el tipo de sentencia que es, y prepara los datos, activa los flags, o realiza las llamadas necesarias para que la sentencia se procese de la manera adecuada.
Inputs	Línea con la sentencia. Índice de la línea.
Outputs	Ninguno.

parser	
Descripción	Método que analiza cada línea identificando el tipo de sentencia que es, y prepara los datos, activa los flags, o realiza las llamadas necesarias para que la sentencia se procese de la manera adecuada.
Inputs	Línea con la sentencia. Índice de la línea.
Outputs	Ninguno.

get_contador_programa	
Descripción	Método que devuelve el número total de pasos que ha dado la ejecución hasta llegar al final del programa.
Inputs	Ninguno.
Outputs	Número total de pasos dados.

get_linea_contador	
Descripción	Método que devuelve el índice de la línea que se ha ejecutado en un paso determinado.
Inputs	Número de paso a devolver.
Outputs	La dirección de la línea en el código.

get_tam_variables_paso	
Descripción	Método devuelve un Map con los tamaños de las variables en un paso determinado.
Inputs	Número de paso a devolver.
Outputs	Map con clave el nombre de la variable y contenido el tamaño.

get_valor_variables_paso	
Descripción	Método devuelve un Map con los valores de las variables en un paso determinado.
Inputs	Número de paso a devolver.
Outputs	Map con clave el nombre de la variable y contenido el valor, la dirección a donde apunta, o el array que lo forma.

get_tam_variables_local_paso	
Descripción	Método devuelve un Map con los tamaños de las variables locales en un paso determinado.
Inputs	Número de paso a devolver.
Outputs	Map con clave el nombre de la variable y contenido el tamaño.

get_valor_variables_local_paso	
Descripción	Método devuelve un Map con los valores de las variables locales en un paso determinado.
Inputs	Número de paso a devolver.
Outputs	Map con clave el nombre de la variable y contenido el valor, la dirección a donde apunta, o el array que lo forma.

get_tam_estructuras_paso	
Descripción	Método devuelve un Map con los tamaños de las estructuras en cada caso.
Inputs	Número de paso a devolver.
Outputs	Map con clave el nombre de la estructura y contenido el tamaño.